Storm 下基于最佳并行度的贪心调度算法 *

熊安萍，段杭彪，蒋亚雄

(重庆邮电大学 计算机科学与技术学院, 重庆 400065)

摘　要： 开源分布式实时计算框架 Storm 在互联网、金融、电子商务等领域得到了广泛应用。Storm 默认采用轮询的调度策略，且依赖用户对 Topology 任务的并行度配置，当配置不合理时依然会造成 Topology 处理时延增大、吞吐量降低等问题。针对该问题，提出了一种 Storm 下基于最佳并行度的贪心调度算法，调度时先求解 Topology 任务中各组件的最佳并行度，再采用贪心策略进行调度，以最小化节点间的网络通信开销。通过与默认调度算法、线上调度算法和热边调度算法进行实验比较，结果表明算法能够有效降低 Storm 处理时延，提高系统吞吐量和资源利用率。

关键词： 实时计算；Storm；最佳并行度；贪心策略；调度算法

中图分类号： TP311　　doi: 10.3969/j.issn.1001-3695.2017.11.0788

Greedy scheduling algorithm based on best parallelism in Storm

Xiong Anping, Duan Hangbiao, Jiang Yaxiong

(*College of Computer Science & Technology*, *Chongqing University of Posts & Telecommunications*, *Chongqing* 400065, *China*)

Abstract: Open-source distributed real-time computing framework Storm in the Internet, finance, e-commerce and other fields has been widely used. By default, Storm uses the polling scheduling policy and relies on the user's configuration of Topology tasks in parallel. When the configuration is unreasonable, Storm still causes delays in Topology processing and decreases throughput. To solve this problem, this paper proposes a Greedy Scheduling Algorithm based on best parallelism in Storm. When scheduling, the best parallelism of each component in Topology task is solved first, and then greedy policy is adopted to minimize the network communication between nodes. Compared with the default scheduling algorithm, the online scheduling algorithm and the hot-side scheduling algorithm, the results show that the algorithm can effectively reduce the Storm processing delay and improve the system throughput and resource utilization.

Key words: real-time computing; storm; best parallelism; greedy policy; scheduling algorithm

0　引言

大数据背景下，数据内涵和价值时效性越来越重要，数据的流式特征也越来越显著，而流式计算的重要性也越来越突出[1]。S4、Spark、Storm 等流式计算框架的推出为流式数据实时处理提供了有效途径。Storm 是个实时的、分布式的以及具备高容错的分布式实时系统[2]，以其实时性、高效性[3][4]，被广泛应用于国内外如百度、阿里巴巴、Twitter 等互联网企业。但 Storm 在实际部署应用中[5]，存在诸如 Nimbus 单节点、Tasks 共享 Worker 相互干扰、反压机制、集群节点间的网络通信开销大等问题。其中，高效的调度策略可以有效缓解延迟、吞吐量、负载均衡等问题，成为提升系统效率的关键。

现有调度算法如线上调度算法、热边调度算法，关注了 Storm 中默认调度策略中集群节点间网络通信开销的问题，但在调度中都依赖于用户配置的 Topology 任务，当配置不当时会严重影响 Topology 的执行效率，造成 Tuple 的处理时延增加，吞吐量降低和资源的浪费。

本文的主要贡献如下:a)利用 Topology 中各组件间关联性，在满足后一组件恰好能完全处理完前一组件所生成的数据下，求解到 Topology 中各组件的最佳并行度，来优化用户初始对 Topology 并行度配置不佳的问题;b）基于 Topology 各组件最佳并行度，采用贪心策略在保证节点 CPU 不超载的情况下进行调度，将通信频率高的 Executor 尽可能的分配到同一节点，从而来最小化节点间的网络通信开销，并提高节点的资源利用率。

1　相关工作

Storm 中用户提交的 Topology 任务是一个有向无环图，图中节点表示计算任务又称做 Component（组件）；边描述节点

收稿日期：2017-11-27；**修回日期**：2018-01-12　**基金项目**：重庆市教委科学技术研究项目（KJ1400414）；重庆邮电大学博士启动基金资助项目（A2015-17）；重庆邮电大学自然科学基金资助项目（A2011-29）

作者简介：熊安萍（1970-），女，教授，博士，主要研究方向为高性能计算与信息安全（xiongap@cqupt.edu.cn）；段杭彪（1991-），男，硕士，主要研究方向为大数据与云计算处理；蒋亚雄（1993-），男，硕士，主要研究方向为大数据与云计算处理.

间的数据流向[6]。Component 之间交换的数据单元称做 Tuple。Storm 中有两种类型的组件：

a)Spout: 数据流的源头，读取待处理数据，然后持续不断地发射 Tuple 到 Topology 中。

b)Bolt: 接收 Spout 或其他 Bolt 发送的 Tuple，并按照用户定义好的方式来处理 Tuple，比如将数据持久化到磁盘，执行过滤、聚集等函数操作。

Component 的运行时实例，称做 Task，数量固定且由用户在 Topology 中配置。在 Storm 集群中，每个 Supervisor 可以启动多个 Worker 进程，Worker 进程运行着 Topology 的多个线程，称 Topology 的线程为 Executor。Executor 用于运行组件的实例 Task，不同组件的 Task 只能运行于不同的 Executor 中[7]，其中运行该组件 Task 的 Executor 线程数目，称其为该组件的并行度。

在 Storm 调度算法方面，业界已有许多相关研究：Aniello 等人[8]提出了一种将相互通信频率高的 Executor 调度到同一个 slot 上，从而减少网络通信的调度算法。Xu 等人[9]提出一种基于负载均衡的调度算法，以减少网络通信开销。将 Executor 按照 traffic load 降序排列，然后按序将 Executor 分配到负载最轻的 slot 上，并保证每个 Worker 节点的任务量不会过载。Peng 等人[10]中提出一种在最大化资源利用率下减少网络通信的调度算法。通过将 Task 映射到 Worker 节点，使所有的资源请求都能够得到满足并且各节点不会出现资源过载。Xiong 等人[11]提出 一种基于热边的调度算法，将热边及关联的热边调度到同一节点来降低节点间的网络通信开销。Long 等人[12]结合 Storm 的不同应用场景，如恢复历史调度任务、单节点任务调度、资源需求调度等，对 Storm 的资源分配和任务调度做了优化。Chakraborty 等人[13]提出了一种基于优先级的调度算法，解决集群资源不足时，可能会导致性能下降、甚至完全饿死具有高业务优先级的拓扑任务的问题。Xin 等人[14]提出一种基于分布式 QoS（quality of service）感知的调度算法，使得 Storm 适用于地理信息系统的处理。

以上研究主要从内部通信开销和资源利用率两方面进行优化，均依赖用户配置的 Topology 的并行度，当用户配置不合理时，现有优化调度方案并不能取得很好的优化效果，并可能对集群处理性能造成更严重的影响，本文提出了一种基最佳并行度的贪心调度算法，在兼顾到网络开销的同时，提升处理效率。

2 相关描述和相关定义

在 Storm 中数据流单位为 Tuple。

定义 1 一个 Tuple 所需的处理时间 t_{tuple} 定义为

$$t_{tuple}=t_{rec}+t_{queue}+t_{proc}+t_{generate} \tag{1}$$

其中：t_{rec} 表示 Tuple 由发送节点到处理节点所需的传输时间；t_{queue} 表示 Tuple 在处理节点需要排队等待的时间；t_{proc} 表示 Tuple 的逻辑处理时间；$t_{generate}$ 表示 Tuple 处理完后形成新的 Tuple 的时间。

定义 2 Tuple 的平均处理时间：

$$t_{avg}=\frac{\sum_{i=1}^{N_{tuple}} t_{(i,tuple)}}{N_{tuple}} \tag{2}$$

其中：N_{tuple} 表示一个时间周期内所有组件处理的 Tuple 数量；$\sum_{i=1}^{N_{tuple}} t_{(i,tuple)}$ 表示处理 N_{tuple} 个 Tuple 所花费的时间；

由式（1）（2）得

$$t_{avg}=\frac{\sum_{i=1}^{N_{tuple}}(t_{(i,rec)}+t_{(i,queue)}+t_{(i,proc)}+t_{(i,generate)})}{N_{tuple}} \tag{3}$$

由式(3)知，减少节点间的网络通信，能降低 t_{rec} ，而当 Topology 配置不当时，可能会造成前一组件发送的 Tuple 较快，使得 Tuple 不断累积，造成 t_{queue} 和 t_{proc} 增大，从而使得 Tuple 平均处理时延增大。

定义 3 Topology 中各组件内 Executor 的时效因子 TF，表示当前组件中的 Executor 在时间周期 T_i 内的繁忙程度为

$$TF=\frac{\sum_{j=1}^{Nexecutor} N_{(j,tuple)} * t_{(j,tuple)}}{Nexecutor * T_i} \tag{4}$$

其中：$N_{executor}$ 表示当前组件中 Executor 的并行度；T_i 表示第 i 个时间周期；$N_{(j,tuple)}$ 表示在时间周期 T_i 内当前组件中第 j 个 Executor 所处理的 Tuple 数目；$t_{(j,tuple)}$ 表示在时间周期 T_i 内当前组件中第 j 个 Executor 的 Tuple 平均处理时间。

TF 值越大，则表明该组件的 Executor 越繁忙，即 Tuple 流速过快，Executor 一直忙于处理 Tuple 流，当 TF>0.8 时[15]，可能形成数据流处理瓶颈。

3 基于最佳并行度的贪心调度算法

当 Topology 任务提交到 Storm 到集群中后，Nimbus 节点对其进行调度分配，包括将 Executor 分配到 Worker 上，再将 Worker 分配到 WorkerNode 上，其中 Excutor 之间的通信分为进程内通信，进程间通信和节点间通信。

在首次提交 Topology 时，以 Storm 的默认的调度算法进行调度，同时对该 Topoloyg 进行监控，并将获取的周期内实时运行参数存入数据库。通过对集群进行实时监控，在首次提交运行周期 T 后或超载时，触发新的调度。

重新调度时根据获取到 Topology 的运行数据，求解 Topology 中各组件的最佳并行度，算法伪代码表 2。然后对并行度调整后的 Topology 任务进行分配，采用贪心策略将通信频率高的 Eexecutor 尽量分配到集群中的同一节点上，算法伪代码表 3。算法用到的相关符号参照表 1。

3.1 组件最佳并行度

Storm 中数据流分组包括：

Shuffle Grouping:随机分组，尽量均匀分布到下游 Bolt 中；

Fields Grouping:按字段分组，按数据中 Field 值进行分组，将相同 Field 值的 Tuple 被发送到同一 Task；

All Grouping:广播分组， 每一个 Tuple 都会复制一份给每一个 Bolt 处理；

Global Grouping:全局分组，所有 Tuple 被分配到一个 Bolt 中的一个 Task 中；

None Grouping:不分组，目前等同于 Shuffle Grouping；

Direct Grouping:指定分组，由 Tuple 的发射单元直接决定将 Tuple 发射给哪个 Bolt；

其中以 Global Grouping 为分组方式的组件不需要进行并行度的调整，规定该组件的并行度为 1。

表 1　符号参考表

符号	含意
e_i	{executor$_i$}(i=1,2,3,,,n)
w_i	{workernode$_i$}(i=1,2,3,,,n)
b_k	{bolt$_k$}(k=1,2,3,,,n):Topology 中第 k 个 Bolt 组件。
LE_i	{load-executor$_i$}(i=1,2,3,,,n):executor 的负载
CE_i	{CPU-executor$_i$}(i=1,2,3,,,n):executor 的 CPU 使用率
$R_{i,j}$	{rate$_{i,j}$}(i,j=1,2,3,,,n)相互通信的 e_i 与 e_j 发送 tuple 速率
T	采集周期
LWN_i	{load-workernode$_i$}(i=1,2,3,,,n):集群中节点 workernode$_i$ 的负载，即 CPU 的使用率
$N_{executor}$	{number-executor}:Topology 中 spout 组件的并行度
$N_{(k,executor)}$	{number-executor$_k$}(k=1,2,3,,,n): Topology 中第 k 个组件的并行度，k=1 时，表示 Topology 中 spout 组件，k=2 时，表示第一个 bolt 组件
AE_i	{assign-executor$_i$}(i=1,2,3,,,n):对应 e_i 的调度方案
EC_i	{executor-communication$_i$}(i=1,2,3,,,n):对应 e_i 的通信量，包括 e_i 接受和发送的 Tuple 数

3.1.1 Spout 组件并行度

在一个时间周期开始时，设上一周期 Spout 未处理完的数据量为 $Data_{surplus}$，外部数据到达速率为 V_{come}，组件 Spout 的数据处理速率为 $N_{executor}*V_{proc}$，其中 $N_{executor}$ 是组件 Spout 的 Executor 并行度，V_{proc} 是 Executor 的数据处理速率。为使得到达 Spout 组件的数据都能够被及时处理，因此有：

$$Datasurplus+V_{come}*T_i \le Nexecutor*V_{proc}*T_i \quad (5)$$

由式（4）（5）得

$$Nexecutor=\left\lceil \frac{Datasurplus+V_{come}*T_i}{V_{proc}*T_i} \right\rceil \quad (6)$$

由式（4）可知增大 $N_{executor}$ 能使得 TF 的值变小，降低 Spout 组件的繁忙程度，因此对式（6）中求得的 Executor 并行度向上取整，求得在外部数据到达速率为 V_{come} 时 Spout 组件的最佳并行度。式(6)中 V_{come} 和 V_{proc} 可以通过监控 Spout 组件然后取平均值获得，其中周期 T_i 的选择也需要进行考虑，Storm 中进行一次 Topology 的调整需要一定的时间，因此，这里的时间周期 T_i 不应过短，但也不能太长，周期 T_i 太短，使得调度过于频繁进而增大处理时延，而周期 T_i 太长会导致无法及时对外部变化进行响应，此处 T_i 通过多次实验取得经验值。

3.1.2 Bolt 组件并行度

用户提交的 Topology 任务中，前一个组件输出的 Tuple 数是后一个组件接受的 Tuple 数。定义第 k 个组件中 Executor 的 Tuple 处理速度为 V_{proc}，则组件 k 的 Tuple 处理速率为 $N_{(k,executor)}*V_{proc}$，其中 $N_{(k,executor)}$ 是该组件的 Executor 数量。其前一组件，即第 k-1 个组件的 Executor 的 Tuple 产生速度为 $V_{generate}$，则该组件的 Tuple 产生速度为 $N_{(k-1,executor)}*V_{generate}$，其中 $N_{(k-1,executor)}$ 是该组件 Executor 的并行度。当后一组件能处理前一组件发送的 Tuple 数时，即前一组件产生的 Tuple 将不会被排队，如式(7)所示。

$$N_{(k-1,executor)}*V_{generate}*T_i \le N_{(k,executor)}*V_{proc}*T_i \quad (7)$$

由式（4）（7）得

$$N_{(k,executor)}=\left\lceil \frac{N_{(k-1,executor)}*V_{generate}*T_i}{V_{proc}*T_i} \right\rceil \quad (8)$$

由式（4）知增大 $N_{(k,executor)}$ 会使 TF 的值变小，降低 Bolt 组件的繁忙程度，因此对式（8）中求得的 Executor 并行度向上取整，求得在后一组件能处理前一组件发送 Tuple 速率下第 k 个组件的最佳并行度。式（8）中 $V_{generate}$ 和 V_{proc} 的值可通过实时监控然后取平均值获得。将式（6）代入式（8），求得 Topology 中第一个 Bolt 组件的并行度 $N_{(2,executor)}$，然后同理迭代出 Topology 中其余 Bolt 组件的最佳并行度。

算法 1　求解组件最佳并行度算法

输入：Topology 信息{e₁, e₂, , , eₙ}、$N_{executor}$、$N_{(k,executor)}$，周期 T 内 Topology 的运行数据 $Data_{surplus}$、V_{come}、V_{proc}、$V_{generate}$、$N_{(j,tuple)}$、$t_{(j,tuple)}$。

输出：Spout 组件的最佳并行度 $N_{executor}$，和各 Bolt 组件的最佳并行度 $N_{(k,executor)}$。

1. 获取 Spout 组件中 Executor 的数据平均处理速率。
2. 求解在下一周期 T 内 Spout 需要处理的数据量 $D_{atasurplus}+V_{come}*T_i$。
3. 根据式(6)求得 Topology 中 Spout 组件的最佳并行度 $N_{executor}$。
4. 从第一个 Bolt 组件开始，即 k=2，依次求解 Bolt 组件的并行度。
5. 如果 b_k 的分组方式为 Grobal grouping 则跳至(7)。
6. 根据式(8)求得 Topology 中第 k 个 Bolt 组件的最佳并行度 $N_{(k,executor)}$。
7. 将 b_k 的最佳并行度设置为：$N_{(k,executor)}=1$。
8. 选取 b_{k+1}，重复步骤(5)、(6)，直到求得所有 Bolt 组件的最佳并行度。
9. 输出求解到的 $N_{executor}$、$N_{(k,executor)}$，算法结束。

3.2　基于贪心策略的任务调度

采集周期 T 内，相互通信的 Executor 之间的 Tuple 传输速率 $R_{i,j}$、Executor 的 CPU 使用率 CE_i 和集群中各节点的负载 LWN_i，按 $R_{i,j}$ 进行降序排列。

根据求得的最佳并行度对 Topology 进行调整，并采用贪心策略进行分配调度，如果某个组件的 Executor 并行度需要降低，先对该组件中的 Executor 按 EC_i 进行降序排列，然后去掉 EC_i

低的 Executor，同时去掉和该 Executor 通信的 $R_{i,j}$，假如存在 $R_{k,i}$ 和 $R_{i,j}$，当要去掉 e_i 时，则应同时删除 $R_{k,i}$ 和 $R_{i,j}$。如要增大 Topology 中某个组件的并行度，取该组件中 EC_i 低的 Executor 的通信速率作为该新添加的 Executor 的通信速率，假设存在 $R_{k,i}$ 和 $R_{i,j}$，如添加 e_m，则需添加 $R_{k,m}$ 和 $R_{m,j}$ 作为新的关联关系。

采用贪婪策略对 Topology 进行分配调度时，尽可能将通信频率高的 Executor 分配到集群中的同一节点上，最小化节点间的网络通信开销，但太多的 Executor 分配到同一 WorkerNode 容易造成该节点的超载，使得该节点处理效率降低，Tuple 处理时延增加。因此，在减少节点间通信量和节点负载之间会有一个权衡。

思想是：

1、采集周期 T 内运行数据，将相互通信的 Executor 按 $R_{i,j}$ 进行降序排列。

2、采集周期 T 内运行数据，对集群中各节点按平均 CPU 使用率进行降序排列。

3、采集周期 T 内 Executor 的平均 CPU 使用率，通过 JavaAPI 中的 getThreadCPUTime(threadID)方法来获取每个线程的 CPU 使用率。

4、循环处理 Executor，直到所有的 Executor 分配完。选择通信频率高的 e_i、e_j，然后对 e_i、e_j 进行分配。采用贪心策略，当 e_i ,e_j 都未被分配时，选取集群中负载最低的节点 w_i，然后尝试将 e_i、e_j 放到 w_i 上，如果 w_i 不超载，则 e_i、e_j 分配成功。否则从集群中选取新的负载最低的节点 w_j，然后尝试将 e_i、e_j 分配到节点 w_i 上，直到 e_i、e_j 分配成功。当 e_i、e_j 中有一个 Executor 被分配了，假设 e_i 已被分配到节点 w_i 而 e_j 还未被分配，此时尝试将 e_j 分配到 e_i 所在的节点 w_i 上，如果 w_i 不超载，则 e_j 分配成功，否则重新从集群中选取新的负载最小的节点 w_j，然后尝试将 e_j 分配到节点 w_j，直到成功。

5、调度中满足如下约定：

在同一个节点上将同一个 Topology 的 Executor 放入到同一个 Worker 中，且保证节点不会超载。

本文暂不讨论 Executor，Task 数量配置优化问题。

算法 2　基于最佳并行度的贪心调度算法

输入：集群节点运行数据 LWN_i、AE_i，Topology 信息{e₁, e₂, , , eₙ}、$N_{executor}$、$N_{(k,executor)}$，周期 T 内 Topology 的运行数据 $Data_{surplus}$、V_{come}、V_{proc}、$V_{generate}$、$N_{(j,tuple)}$、$t_{(j,tuple)}$。

输出：AE_i，i=1, 2, 3, , , n，即{e_1, e_2, e_3, , , e_n}对应的分配方案。

1. 开始调度。/*在周期 T 后采用本文调度算法进行重新调度，调度过后，当集群中的节点出现超载或 Executor 出现超载时，再触发新的调度*/

2. 根据表 2 给出的求解组件最佳并行度算法求得 Topology 中 Spout 组件的最佳并行度 $N_{executor}$，Bolt 组件的最佳并行度 $N_{(k,executor)}$。

3. 将 Topology 中对应的组件并行度配置为 $N_{executor}$、$N_{(k,executor)}$。

4. 计算 $R_{i,j}$，将相互通信的 Executor 按 $R_{i,j}$ 数值大小降序排列。

5. 计算 LWN_i，将集群中节点按 LWN_i 数值大小降序排列。

6. 获取集群中负载最小的 LWN_i 节点。

7. 获取通信频率最高的 $R_{i,j}$。

8. 如果在 $R_{i,j}$ 中 e_i 和 e_j 都未被分配，则跳到(9)，否则跳到(12)。

9. 如果 e_i 和 e_j 都能分配到 LWN_i 节点且不超载，则将 e_i 和 e_j 分配到 LWN_i 节点，并将分配方案记录到 AE_i 和 AE_j，否则跳到(10)。

10. 重新获取集群中最新的负载最小的 LWN_{i+1} 节点。

11. 将 e_i 和 e_j 分配到节点 LWN_{i+1}，并将分配方案记录到 AE_i 和 AE_j。

12. 如果 e_i 被分配了，e_j 还未被分配，则跳到(13)，否则跳到(15)。

13. 如果 e_j 能被分配到 e_i 所在的节点且不超载，则将 e_j 分配到 e_i 所在的 LWN_i 节点，并将分配方案记录到 AE_j，否则跳到(14)。

14. 重新获取集群中新的负载最小的 LWN_{i+1} 节点，将 e_j 分配到 LWN_{i+1} 节点，并记录到 AE_j 中，然后跳到(17)。

15. 如果 e_i 能被分配到 e_j 所在的节点且不超载，则将 e_i 分配到 e_j 所在的 LWN_i 节点，并将分配方案记录到 AE_i，否则跳到(16)。

16. 重新获取集群中新的负载最小的 LWN_{i+1} 节点，将 e_i 分配到 LWN_{i+1} 节点，并将分配方案记录到 AE_i 中。

17. 如果{e_1, e_2, e_3, , , e_n}中还有未被分配的 executor，则跳至(7)，直到所有的 executor 被分配。

18. {e_1, e_2, e_3, , , e_n}被分配完，输出分配方案 AE_i，算法结束。

4　实验与结果

使用 Hyperic 的 SIGAR（System Information Gather And Reporter）来读取集群中机器的实时状态，如 CPU、内存。结合 Storm 提供的 Thrift API 采集 Topology 的运行数据，并将自己实现的监控进程以 Deamon 进程方式在后台运行。

实验中利用 Storm 提供的 Pluggable Scheduler[15]来实现本文调度算法。

实验搭建了三台物理机构成的同构集群，每个节点的具体硬件配置：4 核 2.8 GHz CPU，4 GB 内存，50 GB 硬盘和 10 Gbit 网卡。每个节点运行 Ubuntu 12.04 操作系统。其中一个作为主节点运行 Nimbus 进程，负责资源分配和任务调度；从节点负责接受 Nimbus 分配的任务,启动和停止属于自己管理的 Worker 进程。同时还搭建了 Zookeeper[16] 集群，它们始终处于运行状态。

实验中使用经典的 WordCount 应用，Topology 包括一个 Spout 和两个 Bolt，Spout 从 Redis 中读取数据，形成 Tuple 发送给后面的 Bolt；第一个 Bolt 对 Tuple 中的行记录进行分词，然后形成新的 Tuple 发送给后继 Bolt；第二个 Bolt 用来对词进行统计[17]。

1)实验 1　以一定速率向 Redis 中写入数据，然后 Spout 从 Redis 中读取数据。在将 Topology 提交到集群运行后，本文调度算法在周期 T 后会对 Topology 进行重新调度，统计从第 10 分钟开始到 20 分钟的的运行数据，并多次实验取平均值。

图 1 显示的是默认(default)调度算法、线上调度(online)算法、热边调度算法和本文调度算法之间处理时延对比。online 和热边调度算法与默认调度算法相比，主要优化了集群节点间的

网络通信开销，降低了处理时延。本文调度算法相对 online 调度算法和热边调度算法是基于 Topology 的最佳并行度来进行调度，采用贪心策略进行任务调度，最大化的减少了节点间的网络通信，实验表明本文调度算法比默认调度算法在整体处理时延上降低了约 25%，比 online 和热边调度算法在整体处理时延上降低了约 10%。

图 2 显示的是 4 种调度算法之间吐出量的差别，与默认调度算法、线上调度算法和热边调度算法相比本文调度算法吐出量更高。

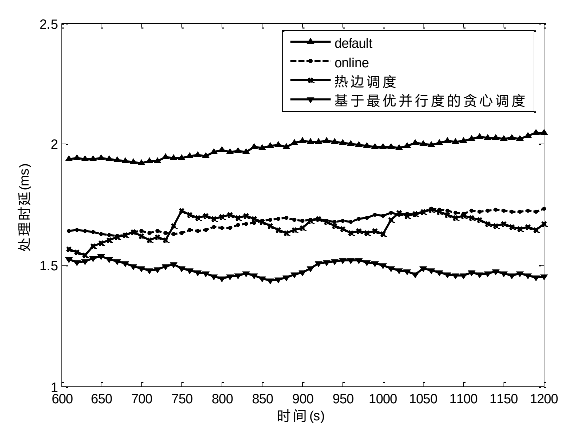


图 1　处理时延对比

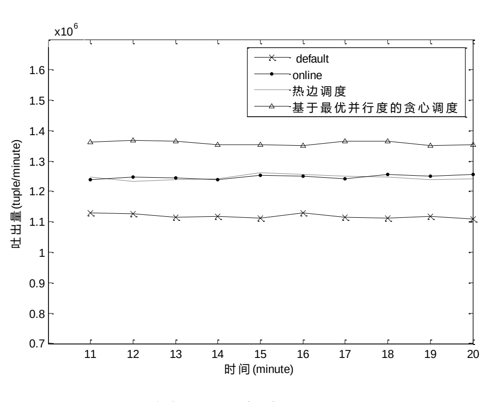


图 2　吞吐量对比

将 default、online 和热边调度算法中 Topology 的并行度分别配置为本文算法得到的最佳并行度，再分别提交 Topology 任务，运行结果图 3 和图 4。由图 3 知，在 Topology 配置为最佳并行度下默认调度算法、线上调度算法和热边调度算法的处理时延整体有所下降，更加接近本文调度算法，同时看出基于最佳并行度的贪心调度算法在时效性上最佳。

图 4 显示的是 Topology 任务被分配到的节点的平均 CPU 使用率。实验结果表明基于最佳并行度的贪心调度比线上调度和热边调度的节点平均 CPU 使用率高 36%左右，比默认调度的节点平均 CPU 使用率高 45%左右。这是由于在默认调度时采用轮询的方式进行任务调度，会将任务平均地分配到集群中的可用节点上，而线上调度和热边调度是每次从集群中去选取负载最低的节点进行分配，而本文算法采用贪心策略在不超过节点负载的情况下会尽可能的将相互通信频率高的 Executor 分配到同一节点上，这样能减少 Topology 任务分配的节点，提高 Topolgy 任务所分配节点的 CPU 使用率，从而使得节点的资源利用率更高。

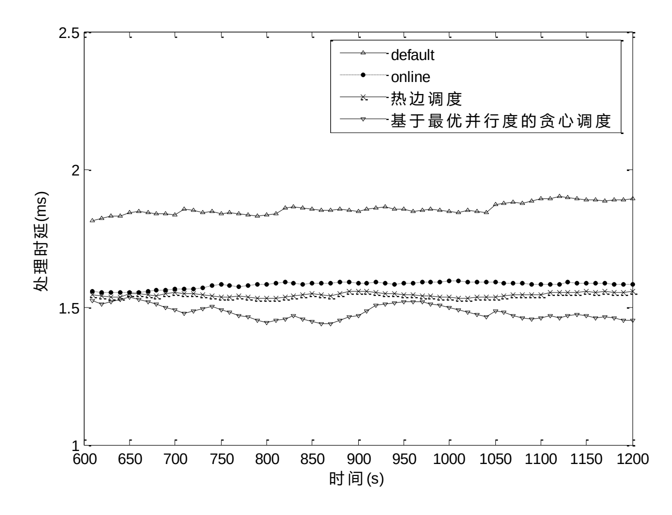


图 3　处理时延对比

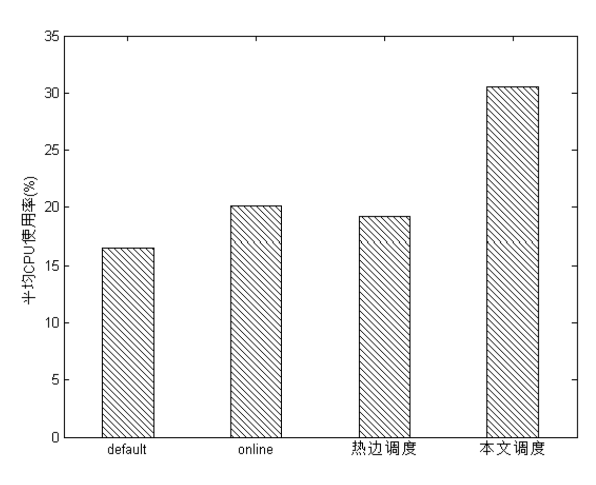


图 4　平均 CPU 使用率对比

b)实验 2　不同数据速率下的平均处理时延

实验中包括五组对比实验，分别以 1000 条/s、2000 条/s、3000 条/s、4000 条/s、5000 条/s 的速率向 Redis 中写入数据，然后 Spout 从 Redis 中读取数据。图 5 显示的是默认调度算法和本文调度算法分别在不同数据输入速率下的平均处理时延。

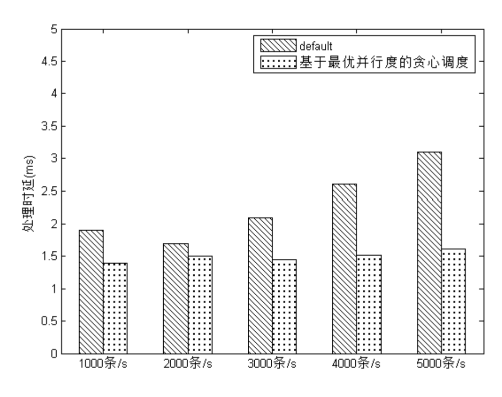


图 5　处理时延对比

Topology 中后端数据处理组件 Bolt 的处理速度较慢，当写入数据速率较大时，如果 Topology 并行度配置不当，默认调度算法容易使数据发送端发送的数据产生累积，造成处理时延的增大。实验结果表明本文调度算法比默认调度算法时效性更好，处理时延更低，并且在处理不同速率数据流时更稳定，效果更好。

5 结束语

本文提出的基于最佳并行的贪心调度算法，首先考虑了 Topology 中组件并行度对调度的影响，然后求解出最佳并行度，并在最佳并行度上采用贪心策略进行调度，和默认调度算法、线上调度算法、热边调度算法相比，本文改进的调度算法，在整体性能上有所提升。在采用贪心策略进行调度时，由于最小化节点间的网络通信，可能会出现负载不均衡的现象。下一步的工作重点将基于本文算法关注调度过程中集群的负载均衡问题。

参考文献：

[1] 孙大为，张广艳，郑纬民. 大数据流式计算：关键技术及系统实例 [J]. 软件学报, 2014, 25 (4): 839-862.

[2] 孟小峰，慈祥. 大数据管理：概念、技术与挑战 [J]. 计算机研究与发展, 2013, 50 (1): 146-169.

[3] Lim L, Misra A, Mo T. Adaptive data acquisition strategies for energy-efficient, smartphone-based, continuous processing of sensor streams [J]. Distributed and Parallel Databases, 2013, 31 (2): 321-351.

[4] 王铭坤，袁少光，朱永利，等. 基于 Storm 的海量数据实时聚类 [J]. 计算机应用, 2014, 34 (11): 3078-3081.

[5] 王润华，毋建军，侯佳路. 分布式实时计算引擎——Storm 研究 [J]. 中国科技信息, 2015 (6): 68-69.

[6] Ghaderi J, Shakkottai S, Srikant R. Scheduling storms and streams in the cloud [C]// Proc of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. New York: ACM Press, 2015: 439-440.

[7] Toshniwal A, Taneja S, Shukla A, *et al.* Storm@twitter [C]// Proc of ACM SIGMOD international. Conference on Management of Data. New York: ACM Press, 2014: 147-156.

[8] Aniello L, Baldoni R, Querzoni L. Adaptive online scheduling in storm [C]// Proc of ACM International Conference on Distributed Event-Based Systems. New York: ACM Press, 2013: 207-218.

[9] Xu J, Chen Z, Tang J, *et al.* T-Storm: Traffic-Aware Online Scheduling in Storm [C]// Proc of IEEE International Conference on Distributed Computing Systems. 2014: 535-544.

[10] Peng B, Hosseini M, Hong Z, *et al.* R-Storm: resource-aware scheduling in Storm [C]// Proc of MIDDLEWARE Conference. New York: ACM Press, 2015: 149-161.

[11] 熊安萍，王贤稳，邹洋. 基于 Storm 拓扑结构热边的调度算法 [J]. 计算机工程, 2017, 43 (1): 37-42.

[12] Long S, Rao R, Miao W, *et al.* An improved topology schedule algorithm for storm system [C]// Proc of Asia-Pacific Conference on Computer Science and Applications. [S. l.] : CRC Press, 2015: 187.

[13] Chakraborty R, Majumdar S. A priority based resource scheduling technique for multitenant storm clusters [C]// Proc of International Symposium on PERFORMANCE Evaluation of Computer and Telecommunication Systems. 2016: 1-6.

[14] Xin Q, Yao X. Distributed QoS-aware scheduling in cognitive radio cellular networks [C]// Proc of International Conference on Network and Information Systems for Computers. Washington DC: IEEE Computer Society, 2015: 106-110.

[15] Apache Storm [EB/OL]. [2017-11-25]. http://storm. apache. org/

[16] Apache ZooKeeper [EB/OL]. [2017-11-25]. https://zookeeper. apache. org/

[17] 李川，鄂海红，宋美娜. 基于 Storm 的实时计算框架的研究与应用 [J]. 软件, 2014 (10): 16-20.